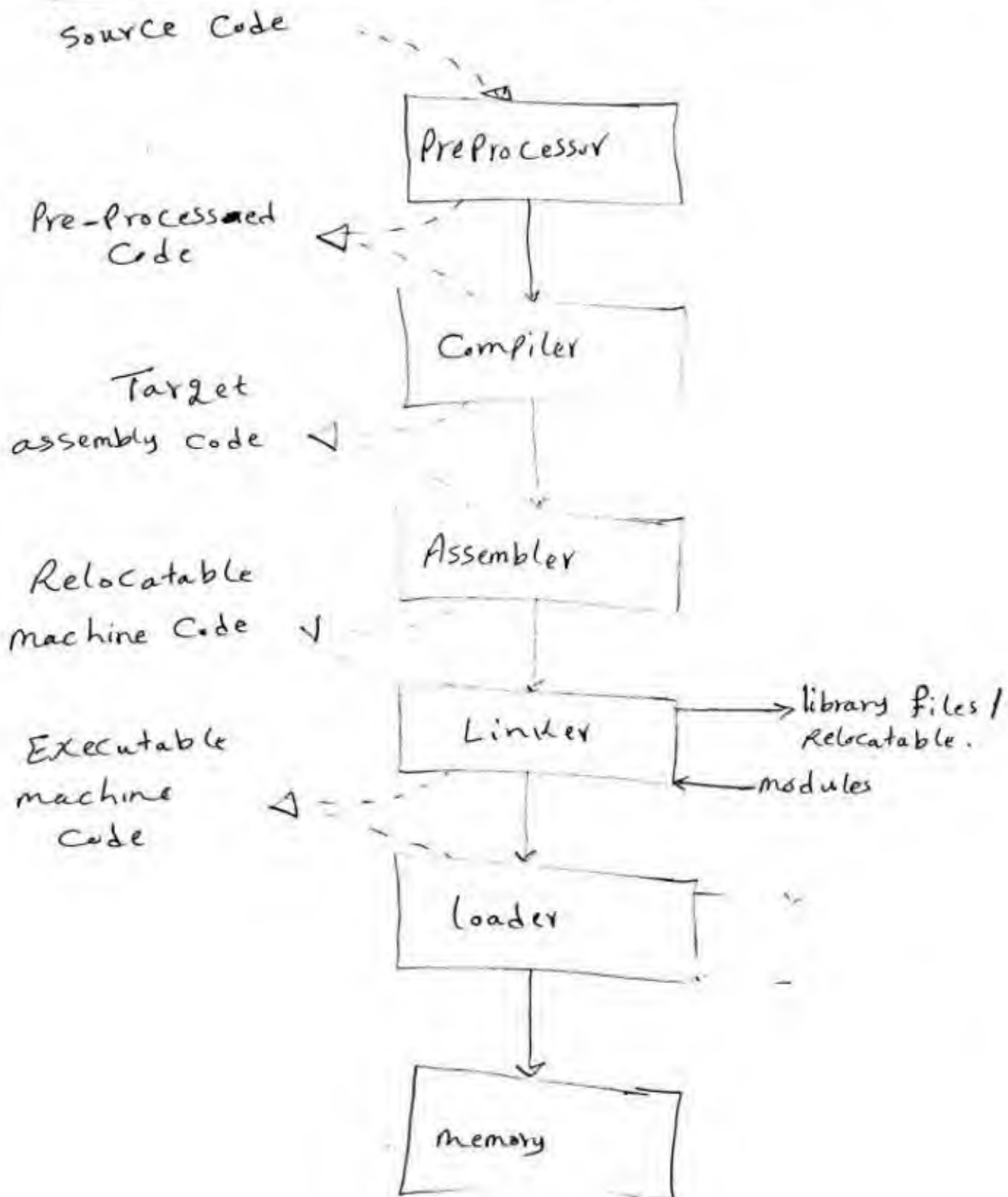


Lec 1

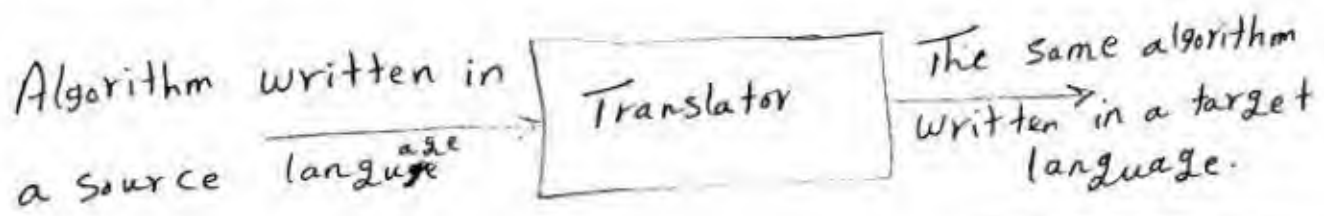
→ Language Processing System:-



→ Translators

* do the same functions of compilers.

* A program which accept a textual representation of an algorithm expressed in a source language, and produce primary output a representation of the same Algorithm. expressed in another Language.



Types of translators

→ Pre Processor

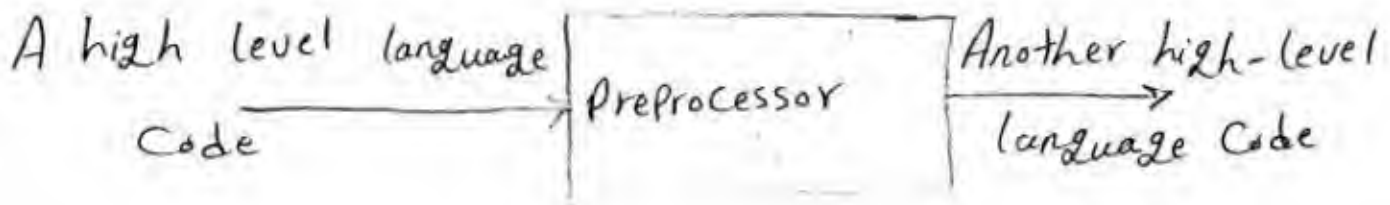
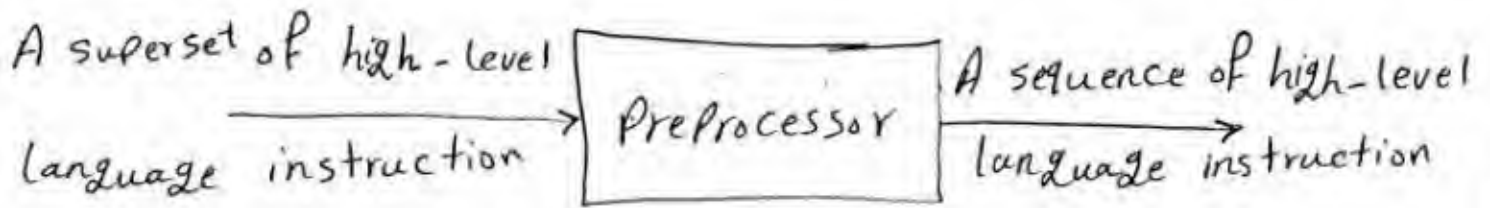
* Considered as a part of compiler.

* Produces ~~input~~ input for compilers.

* deals with macro-Processing, File inclusion, language extension, ...etc.

← يأخذ ال (micro instructions) ويحجزها بذاكرة.

* It can translate from high level language to another high level language.



Notes

→ Some Compilers can give us executable code (ready) or assembly code (need some operations to be ready)

→ Translator has to let two programs do the same function to translate between them.

Types of translators "continued"

② Assembler :-

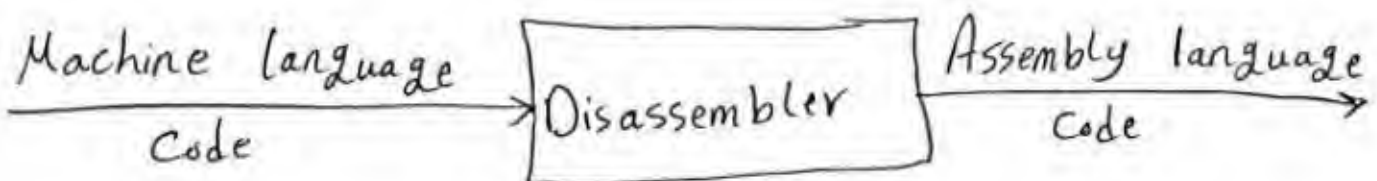
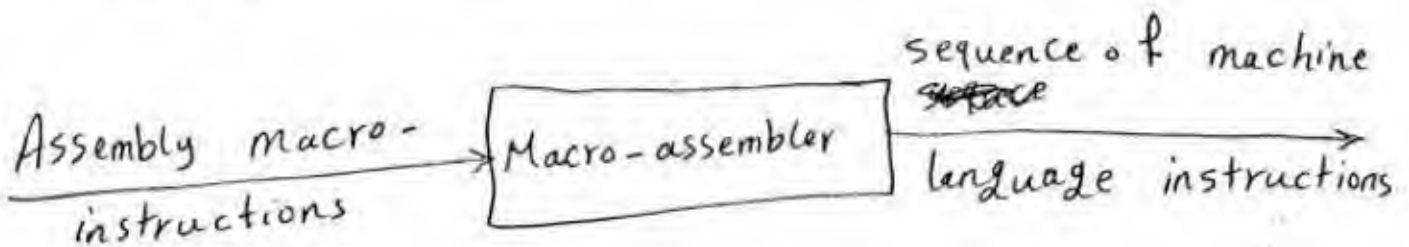
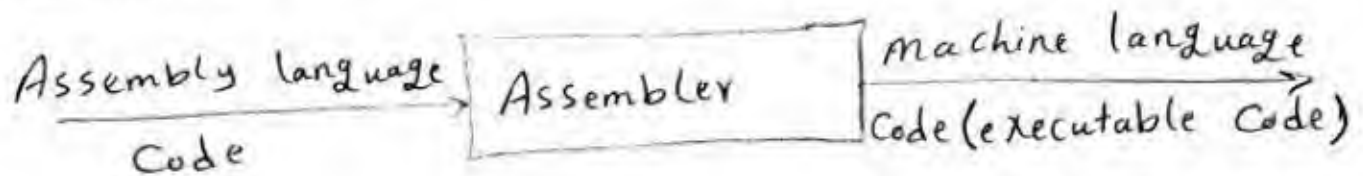
* translates assembly language Programs into machine code.

* It is (one-to-one)

(one instruction) ← (assembly instruction)

→ (machine) ← (O/P) ← (run) ←

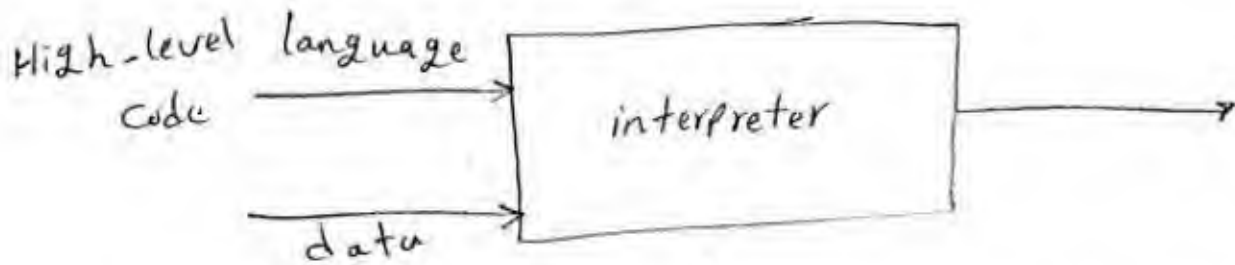
* It's o/p called object file, which contains combination of machine instructions.



Types of translators "Continued.."

3) interpreter:-

→ translates high-level language into low-level machine Language.



interpreter

→ translates program one statement at a time.

→ No intermediate object code is generated hence are memory efficient.

→ less ~~amount~~^a amount of time to analyze source code

→ overall execution time is slower.

→ continue translating until program until first error is met, it stops.

→ Hence debugging is easy.

Compiler

→ scan the entire program & translates it as a whole into machine code

→ Generates intermediate object code which requires linking hence requires more memory.

→ large amount time to analyze source code.

→ overall execution time is comparatively faster

→ Generates error message only after scanning whole program.

→ debugging is comparatively hard.

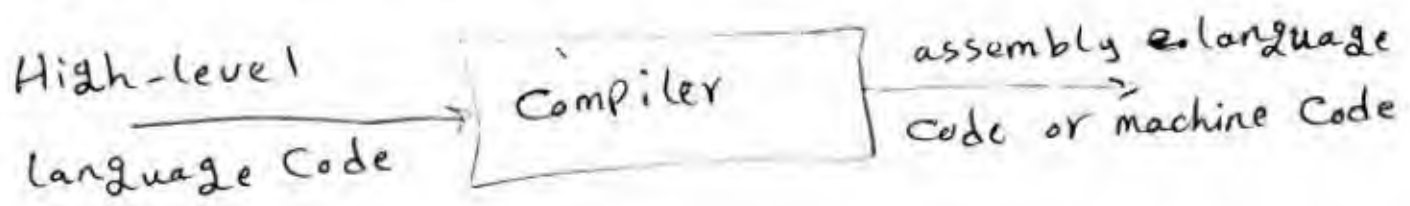
→ Programming language like Python ~~use~~ use interpreter.

→ Programming language like C, C++ ~~use~~ use Compilers

Compilers

→ it is a program can read a program in one language - source language - and translate it into an equivalent program in another language - the target language.

→ report any errors in source program that it detects during the translation process.



→ classified with consideration of ^{comp} instruction

a) Single Pass b) Multiple Pass

c) Load/Go.

→ classified with consideration of optimization:-

a) global

b) Local.

* Cross Compiler

→ Compiler that runs on Platform (A) and is capable of generating executable code for Platform (B).

* Source-to-Source Compiler:-

→ A compiler takes the source code of one programming language and ~~translates~~ translates it into source code of another programming language.

* A Decompiler:-

→ Program translates from low-level language (machine language) to a ~~high~~ higher-level language.

* Preprocessor is similar to source-to-source compiler in the definition.

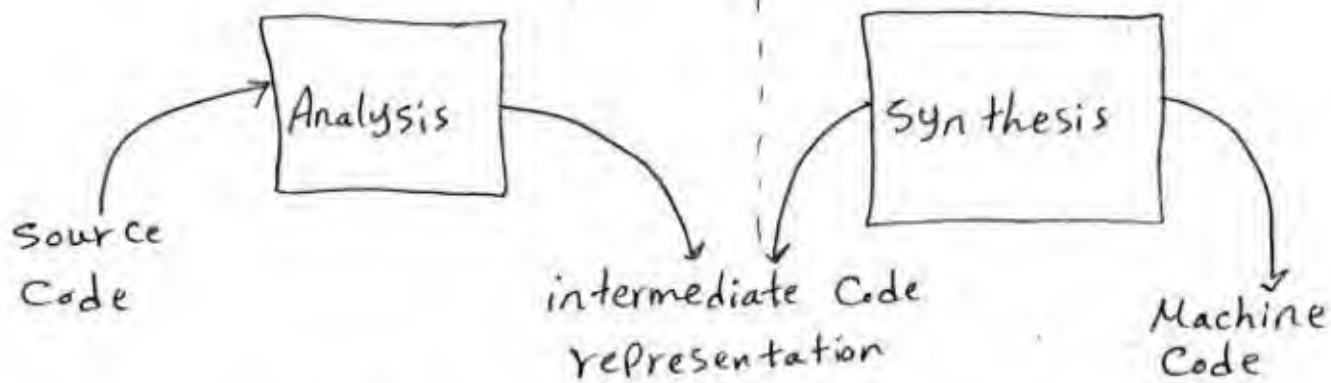
→ Compilers ~~is~~ are classified with consideration of function:-

a) debugging

b) optimization.

Compiler Architecture

Front-end Back-end




* Analysis Phase

- Known as front-end of the compiler.
- it reads source program, then divides it into core parts, and then check for lexical, grammar and syntax errors.
- Analysis goes in three parts
 - a) lexical (scanner)
 - b) syntax.
 - c) semantic.
- it generates an intermediate representation of the source program and symbol table which should be fed to synthesis phase as input.

* Synthesis Phase

- Known as the back-end of the Compiler.
- generates the target Program with the help of intermediate source code representation and symbol table.

Front end  Analysis
intermediate code representation.

backend (back end) ، (front end) القياس ما بين
هو علاقة بال (machine).

back-end → if we work with local optimization.

~~synthesis~~ machine independent → global optimiser
machine dependent → local " .

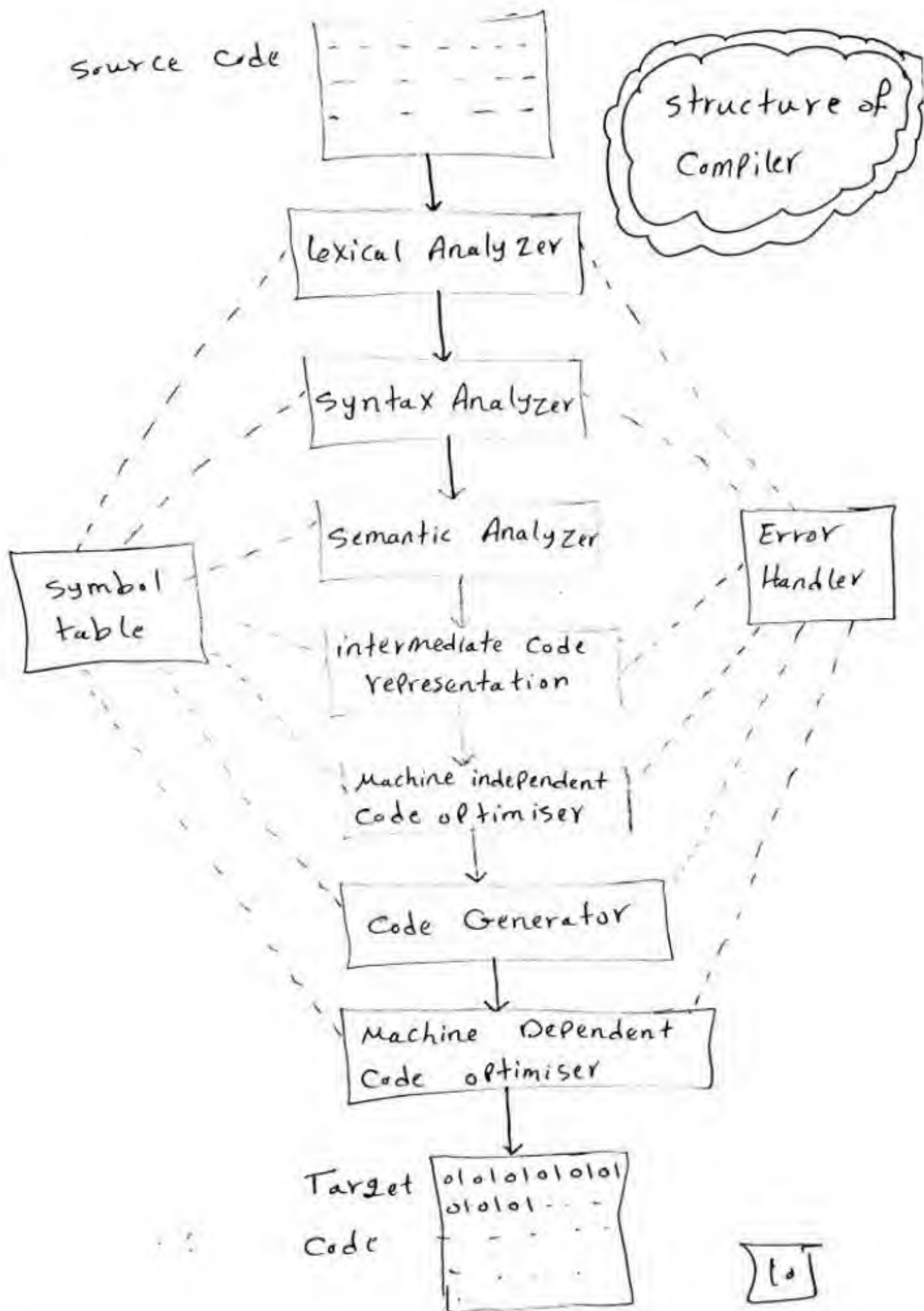
* Phases we will study more

* Lexical

* syntax

* Code Generator.

* machine dependent Code optimiser.



* lexical analysis "scanner"

→ First Phase of Compiler.

→ works as a text scanner.

→ scans the source code as a stream of characters and converts it into meaningful lexemes.

→ represents these lexemes in the form of tokens as:

< token-name, attribute-value >

→ lexemes must be meaningful as a language.

← تری ال (Command) فی معنی ال (Preprocessor) من پیشہ .

→ takes statement by statement.

← ال tokens فی معنی (word) لکھ فی البرنامج .

This is a sentence.

→ is not trivial. Consider:

ist his asente nce.

→ Lexical analyzer divides Program text into "words" or "tokens".

if $x == y$ then $Z = 1$; else $Z = 2$;

Syntax Analysis or Parsing

→ it takes the token produced by lexical analysis as input and generates a parse tree.

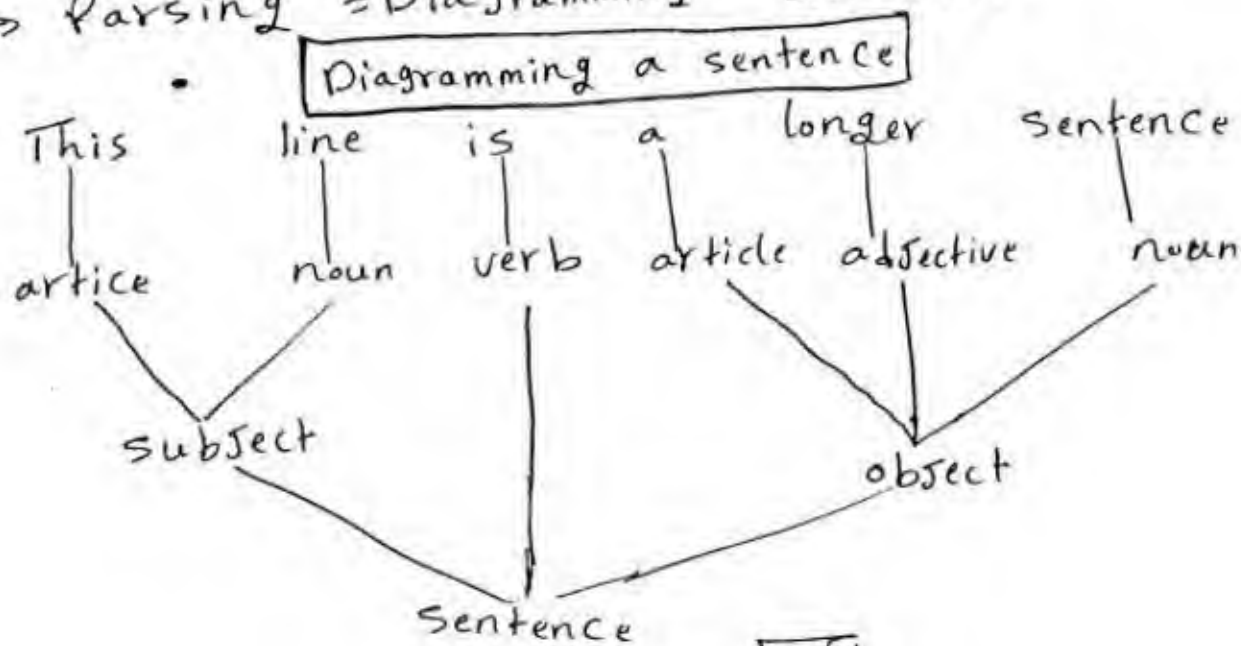
→ check source code for errors.

→ give us parse tree or atoms as o/p.

→ search for the sentence's structure.

→ once words are understood, next step is to understand sentence structure.

→ Parsing = Diagramming Sentences

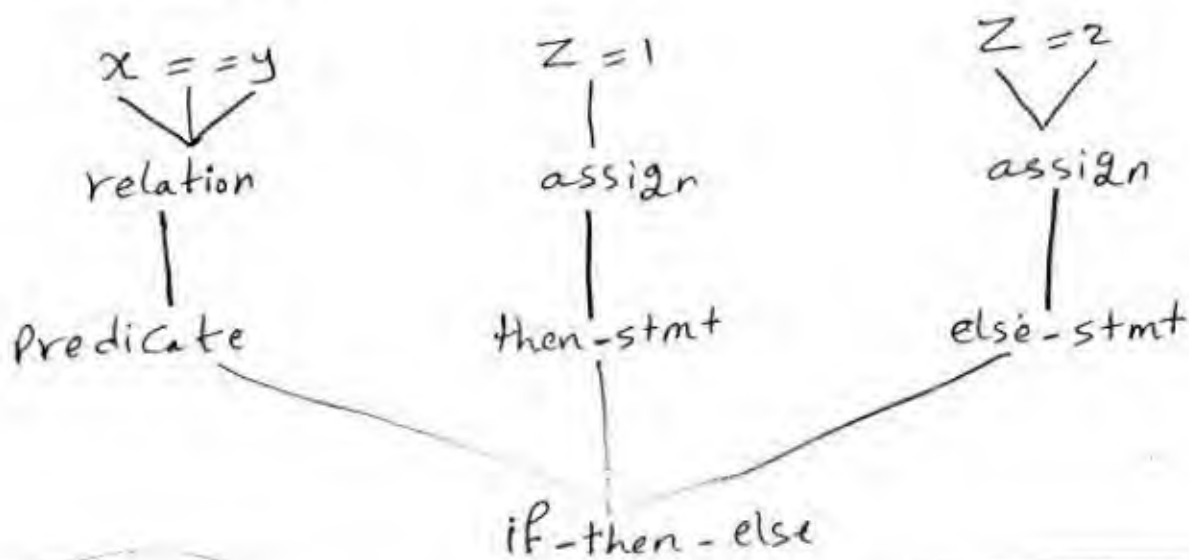


→ Parsing Program expressions is the same:-

Consider:

if $x == y$ then $z = 1$; else $z = 2$;

Diagrammed



Semantic Analysis

→ once sentence structure is understood, we can try to understand "meaning".

→ Compilers perform limited analysis to catch inconsistencies.

→ Semantic checks whether the parse tree constructed follows the rules of language.

EX

Jack said Jerry left his assignment at home

→ What does "his" refer to? Jack or Jerry?

Even worse:

Jack said Jack left his assignment at home?

How many Jack are there?

Which one left the assignment?

→ Semantic on Programming

```
{  
  int Jack = 3;  
}  
{  
  int Jack = 4;  
  cout <<  
    Jack;  
}
```

→ This C++ Code
Prints "4"; inner
definition is used.

→ Compilers perform many semantic checks
besides variable bindings.

Ex: Jack left her homework at home

→ A "type mismatch" between her and Jack.

Optimization

- Can be assumed as something that removes unnecessary code lines.
- arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory)
- Automatically modify programs so that they:
 - * Run faster.
 - * use less memory.
 - * in general, conserve some resource.

Ex

$X = Y * 0$ is the same as $X = 0$

Code Generation

- usually produces assembly code.
- takes the optimized representation ~~code~~ of intermediate code and maps it to the target machine language.
- translates the intermediate code into a sequence of re-locatable machine code.

→ ~~Sequence~~

→ A translation into another language
* analogous to human translation.

Intermediate language

→ It represents a program for some abstract machine.

→ It is in between the high-level language and the machine language.

→ Intermediate code should be generated in such a way that it makes it easier to be translated into target machine language.

Note

Compiler Function

→ translate from high level language to low level language.

→ handling for error.

→ build symbol table.

Symbol table

- data-structure maintained throughout all the phases of Compiler.
- All the identifiers' names along with their types are stored here.
- makes it ~~easy~~ easier for compilers to quickly search the identifier record and retrieve it.
- used for scope management.

— يأخذ مكانه في ال (memory)

— أي (variable) يخزنه في (symbol table)

يحجز مكانه في الذاكرة ثم يتم تخزين ال (symbol table)

في الذاكرة أي أنه يأخذ مكانه في ال (memory).